Peridigm

Peridigm Development Guide

For Peridigm versions $\geq 1.4.1$

Martin Rädel and Christian Willberg

DLR Deutsches Zentrum für Luft- und Raumfahrt
German Aerospace Center

DLR German Aerospace Center
Composite Structures and Adaptive Systems
Structural Mechanics
Dr. Tobias Wille

38108 Braunschweig
Germany

Tel: +49 (0)531 295-3701
Fax: +49 (0)531 295-3702
Web: http://www.dlr.de/fa/en

Martin Rädel and Christian Willberg

Tel: +49 (0)531 295-2048
Fax: +49 (0)531 295-2232
Mail: martin.raedel@dlr.de

Repository
This document is part of the PeriDoX repository [1].
The complete repository can be found at:
https://github.com/PeriDoX/PeriDoX

Citing
When citing this document, please reference the following:
Martin Rädel, Christian Willberg, Peridigm Users Guide, DLR-IB-FA-BS-2018-23, DLR Report, 2018, in PeriDoX, DOI: 10.5281/zenodo.1403015

Contents

List of Figures

List of Tables

List of Symbols

1. About

1.1. Scope

This document is supposed to be a documentation how to develop in *Peridigm*.

2. *Peridigm* - Development Guide

2.1. Program structure

The whole analysis is organized in

```
./src/core/Peridigm.cpp
```

Here, the gobal data structure, the different solvers and the model evaluation calls can be found.

2.1.1. Remark on data structure

In Peridigm bonds exists only as neighbor. As a consquence the information of an bond P_1P_2 is not necessarily equal to P_2P_1, cf. Figure 2.1. Let assume P_1 has a damage model and P_2 not. As result in Peridigm the bond P_1P_2 can be deleted, but not the bond P_2P_1.

2.1.2. Numbering

There is no unique numbering within Peridigm. The nodenumber is the offset value to a pointer address. Therefore, a cross reference to another node is hard to create. The general structure is the following

It must be noted that this structure exists also if multiple cores are used. However, only parts of the loop are used in that case. The bond numbering is counted continuously within the inner loop j. Based on this structure, it is clear that the bond ij is not equal to ji.

2.1.3. Model evaluator

The evaluation of algorithm 1 is done in Peridigm.cpp

```
modelEvaluator->evalModel(workset);
```

```
initialization;
updateDisplacementsToBlocksAndCores;
for blockID ← 1 to n_blocks do
    for i ← 1 to n_nodes do
        for j ← 1 to n_neighbors do
            calculateDamages;
            bondNumber++;
        end
    end
    bondNumber = 0; for i ← 1 to n_nodes
     do
        for j ← 1 to n_neighbors do
            calculateBondForces;
            bondNumber++;
        end
    end
    if (contact) calculateContact;
end
synchronizeForcesInGlobalVector;
timeIntegrationInGlobalVector;
```

Algorithm 1: Perdigm data structure

This calls evaluation routines. If extra high level routines for evaluation are needed they should be added in

```
./src/core/Peridigm_ModelEvaluator.cpp
./src/core/Peridigm_ModelEvaluator.hpp
```

The main model evaluator routine is split in three main parts. The first part calls the damage model routine. There is a check if a block has damage model or not.

```
damageModel->computeDamage(dt,
                           numOwnedPoints,
                           ownedIDs,
                           neighborhoodList,
                           *dataManager);
```

The second part calls the material.

```
    materialModel->computeForce(dt,
                                numOwnedPoints,
                                ownedIDs,
                                neighborhoodList,
                                *dataManager);
```

The third part calls the contact manager. There is a check if contact is available or not.

```
workset->contactManager->evaluateContactForce(dt);
```

2.1.4. Paralellization

Figure 2.1 shows a simple example how the paralallization works. It has a huge impact to the data exchange and communication between points. The lines between the three points are bonds. It means that P_1 has two neigbors and P_2 or P_3 only one.

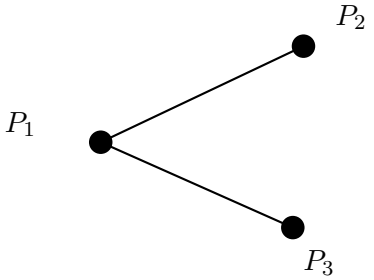

Figure 2.1.: Illustration of core data.

If this problem is parallelized in Peridigm the maximum core number is three. As result each core gets one point as information and the neighbor information as ghost. Ghost means that partially data is synchronized and therefore available for all cores. In case of Peridigm the forces, deformation states and temperatures are synchronized.

Damages are not synchronized. This is important. It means that a damage have to be calculated based on the information at one point using his neighborhood.

As shown in algorithm 1 the synchronization is done outside the model evaluator. Therefore, if a data exchange between the damage and material routines are needed, an additional synchronization has to be added. For further information see subsubsection 2.1.6.1.

2.1.5. Solver

not all solvers support everything

2.1.6. Limitations and Lessons learned

- data between different blocks are only partially available and hard to transfer; e.g. the bond energy is calculated from both sites of the bond. To transfer the energy

the datamanger is used. If multiple blocks exists no consistent datamanager exists and the transfer method does not work.

- working with MPI paralellization have to take into account, that not all data is available. Each node is stored and his neighbors are ghosts. Ghost means that the node information is exchangeable.
- Peridigmis compiled multiple times, make sure that all files are compiled. If new files are included, time stamp differences could make a problem
- field `ownedID` has no meaning; Its been used, but not everywhere
- not all solvers support everything
- work with minimum of 2 cores to avoid synchronization errors

2.1.6.1. Peridigm data structure

Peridigm is structured as shown in algorithm 1. All data is read and stored in blocks. In blocks the material and damage model as well as the horizon is defined. Peridigm reads the data blockwise and store it in the so called data manager.

2.1.6.2. Datamanager

Description

The datamanager allows different types of variables and where to find it. Table 2.1 shows the possible options for the definition. All values are of type double. The NODE and ELEMENT are point data. In ParaViewthe data is handled differently. NODE is exported as POINT data and ELEMENT data is exported as cell data.

The data is stored via MPI at each core. There is so called "ghost" data which is the connection between two computer cores and is the only data which exists at both computer cores.

Table 2.1.: Data types

Types	Time	Length	Description
ELEMENT	CONSTANT, TWO_STEP	SCALAR, VECTOR, TENSOR	data stored as cell data for ParaView
BOND	CONSTANT, TWO_STEP	SCALAR	connection between two points (12 and 21 are separate entries)
NODE	CONSTANT, TWO_STEP	SCALAR,VECTOR, TENSOR	points

To create a datamanager field the following commands have to be defined.

```
m_damageFieldId = fieldManager.getFieldId(PeridigmNS::PeridigmField::
    ELEMENT, PeridigmNS::PeridigmField::SCALAR, PeridigmNS::PeridigmField
    ::TWO_STEP, "Damage");
m_fieldIds.push_back(m_damageFieldId);
```

The different time values are

```
PeridigmField::STEP_NP1, PeridigmField::STEP_N, PeridigmField::STEP_NONE
```

Peridigm checks if the datafield exists and if not creates it with the defined id. To get the data you have to call

```
double *bondDamage;
dataManager.getData(m_bondDamageFieldId, PeridigmField::STEP_NP1)->
    ExtractView(&bondDamage);
```

The value you will get are a pointer with different lengths. The lengths are given on Table 2.1 and given in Table 2.2.

Table 2.2.: Size of data types

Types	Length	Factor	Description
ELEMENT	number of points n_{points}	-	data stored as cell data for ParaView
BOND	number of points n_{points}	-	
NODE	number of bonds n_{bonds}	-	
SCALAR		1	
VECTOR		2	
TENSOR		9	
CONSTANT	-	1	
TWO_STEP		2	

Example

The size of the damage field is

$$size = ELEMENT \cdot SCALAR \cdot TWO\_STEP = n_{points} \cdot 1 \cdot 2 \tag{2.1}$$

Data synchronization

The data synchronization can be done in

```
./src/core/Peridigm.cpp
./src/core/Peridigm.hpp
```

Examplery, this will be shown for the damageModelField. This field is used to synchronize data for the energy criterion. To do a synchronization the following steps has to be done. In Peridigm.hpp you have to define a field id for your datamanager field

```
// field ids for all relevant data
int damageModelFieldId;
```

and you have to define a global vector

```
//! Global vector for damage model data
Teuchos::RCP<Epetra_Vector> damageModelVal;
```

In a second step you create a datamanager field in Peridigm.cpp as

```
damageModelFieldId = fieldManager.getFieldId(PeridigmField::NODE,
   PeridigmField::VECTOR, PeridigmField::TWO_STEP, "Damage_Model_Data");
auxiliaryFieldIds.push_back(damageModelFieldId);
```

Due to the low number of comments the search tag to find the position is

```
// Create field ids that may be required for output
```

The datamanager field is the interface to the core data. In the next step the synchronization vector has to be defined. The search tag is

```
// Create mothership vectors
```

Mothership in this context mean global data off all cores. This data includes for example the forces or displacements. Dependent on the synchronization data the user could choose between oneDimensionalMap, threeDimensionalMap and nDimensionalMap. Two modifications have to be done. First the number of fields has to be adapted. In the example from ten

```
threeDimensionalMothership = Teuchos::rcp(new Epetra_MultiVector(*
   threeDimensionalMap, 10));
```

to eleven

```
threeDimensionalMothership = Teuchos::rcp(new Epetra_MultiVector(*
   threeDimensionalMap, 11));
```

Next the field itself has to be defined.

```
damageModelVal = Teuchos::rcp((*threeDimensionalMothership)(9), false);
   // Damage Model data which has to be synchronized
```

The name of the vector must be the same as the defined one in the Peridigm.hpp. Only then the vector can be used anywhere in Peridigm.cpp.

To synchronize the data the following two loops have to be used at the correct positions. The first loop is for the core data collection.

```
 for(blockIt = blocks->begin() ; blockIt != blocks->end() ; blockIt++){
        scratch->PutScalar(0.0);
        blockIt->exportData(*scratch, damageModelFieldId, PeridigmField::
   STEP_NP1, Add);
        damageModelVal->Update(1.0, *scratch, 1.0);
```

```
}
```

The data is stored within the scratch field and then added to the mothership vector. The mothership vector data is updated as

$$mothership = \sum_{i}^{n_{cores}} v_i^{scratch} \tag{2.2}$$

Taking the simple three point problem of Figure 2.1 should illustrate the synchronization of the force vector

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{pmatrix} = \begin{pmatrix} f_1^{core_1} \\ f_2^{core_1} \\ f_3^{core_1} \\ f_4^{core_1} \\ f_5^{core_1} \\ f_6^{core_1} \end{pmatrix} + \begin{pmatrix} f_1^{core_2} \\ f_2^{core_2} \\ f_3^{core_2} \\ f_4^{core_2} \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} f_1^{core_3} \\ f_2^{core_3} \\ 0 \\ 0 \\ f_5^{core_3} \\ f_6^{core_3} \end{pmatrix} \tag{2.3}$$

As one can see, the summation leads to the transfer of information from the neighbor to the point. **If this is not necessary the neighbor data must be zero** to avoid unreasonable results.

To map the values back to the cores the following loop have to be used.

```
for(blockIt = blocks->begin() ; blockIt != blocks->end() ; blockIt++)
{
  blockIt->importData(*damageModelVal, damageModelFieldId,
PeridigmField::STEP_NP1, Insert);
}
```

Remark - Data synchronization

It is not possible to synchronize CONSTANT datatypes.

Data export

The data is exported in a ParaViewreadable result. Not all fields of the data manager could be exported. From my understanding only data which is defined in the Peridigm.cpp data manager and synchronized with the cores could be exported. This can be done in

```
./src/io/mesh\_output/Field.h
```

If an user export should be added the name of the value, defined in the datamanager has to be added in the following list. For example the field DAMAGE could be found in the list.

```
enum Type {
  VOLUME=0,
  DENSITY,
  GID,
  BLOCK_ID,
  PROC_NUM,
  WEIGHTED_VOLUME,
  RADIUS,
  NEIGHBORHOOD_VOLUME,
  NUMBER_OF_NEIGHBORS,
  CRITICAL_TIME_STEP,
  DILATATION,
  DAMAGE,
  CRITICAL_STRETCH,
  E_DP,
  E_DB,
  PLASTIC_CONSISTENCY,
  NORM_DEVIATORIC_FORCE_STATE,
  NUM_NEIGHBORS,
  FLUID_PRESSURE_Y,
  FLUID_PRESSURE_U,
  FLUID_PRESSURE_V,
  FLUX,
  FLUX_DENSITY,
  COORDINATES,
  TANGENT_REFERENCE_COORDINATES,
  DISPLACEMENT,
  CURRENT_COORDINATES,
  VELOCITY,
  ACCELERATION,
  BC_MASK,
  FORCE,
  FORCE_DENSITY,
  CONTACT_FORCE,
  CONTACT_FORCE_DENSITY,
  RESIDUAL,
  BOND_DAMAGE,
  PARTIAL_VOLUME,
  TYPE_UNDEFINED,
  ANGULAR_MOMENTUM,
```

```
  LINEAR_MOMENTUM,
  KINETIC_ENERGY,
  STRAIN_ENERGY,
  STRAIN_ENERGY_DENSITY,
  INTERFACE_PROXIMITY,
  HORIZON
};
```

As reminder not all information is available in all analysis, e.g. the DILATATION exists only Peridynamic solid, the damage index exists only if a damage model is active.

Remark - Export data routines

In

```
./src/compute
```

are several routines which provide the export manager with extra data. For example the deformation gradient of the correspondence formulation is calculated and provided. It is also exported in Field.h. How it works is an open point.

2.2. Additional features

2.2.1. How to read Exodus files with Python

In order to read the output files and evaluate the results indiviually Python has to be installed. Furthermore, the netCDF module for Python is required. The module matplotlib should be installed as well, since Python is able to plot data.
Normally, it makes sense to write an additional output file with Peridigm and store the variables of interest there. The additional output file is defined in the input file as Output2 in the following. We take the input from ViscoplasticNeedlemanFullyPrescribedTension_NoFlaw.xml as an example.

```
<ParameterList name="Compute Class Parameters">
     <ParameterList name="Max Von Mises Stress">
       <Parameter name="Compute Class" type="string" value="Block_Data"/>
       <Parameter name="Calculation Type" type="string" value="Maximum"/>
       <Parameter name="Block" type="string" value="block_1"/>
       <Parameter name="Variable" type="string"
       value="Von_Mises_Stress"/>
       <Parameter name="Output Label" type="string"
       value="Max_Von_Mises_Stress"/>
     </ParameterList>
     <ParameterList name="Min Von Mises Stress">
       <Parameter name="Compute Class" type="string" value="Block_Data"/>
       <Parameter name="Calculation Type" type="string" value="Minimum"/>
       <Parameter name="Block" type="string" value="block_1"/>
       <Parameter name="Variable" type="string"
       value="Von_Mises_Stress"/>
       <Parameter name="Output Label" type="string"
       value="Min_Von_Mises_Stress"/>
     </ParameterList>
</ParameterList>

<ParameterList name="Output1">
        <Parameter name="Output File Type" type="string"
        value="ExodusII"/>
        <Parameter name="Output Filename" type="string"
        value="ViscoplasticNeedlemanFullyPrescribedTension_NoFlaw"/>
        <Parameter name="Output Frequency" type="int" value="50"/>
        <ParameterList name="Output Variables">
          <Parameter name="Volume" type="bool" value="true"/>
          <Parameter name="Displacement" type="bool" value="true"/>
          <Parameter name="Velocity" type="bool" value="true"/>
          <Parameter name="Force" type="bool" value="true"/>
```

```
          <Parameter name="Number_Of_Neighbors" type="bool"
          value="true"/>
          <Parameter name="Hourglass_Force_Density" type="bool"
          value="true"/>
          <Parameter name="Deformation_Gradient" type="bool"
          value="true"/>
          <Parameter name="Left_Stretch_Tensor" type="bool"
          value="true"/>
          <Parameter name="Rotation_Tensor" type="bool" value="true"/>
          <Parameter name="Shape_Tensor_Inverse" type="bool"
          value="true"/>
          <Parameter name="Unrotated_Rate_Of_Deformation" type="bool"
          value="true"/>
          <Parameter name="Cauchy_Stress" type="bool" value="true"/>
          <Parameter name="Unrotated_Cauchy_Stress" type="bool"
          value="true"/>
          <Parameter name="Equivalent_Plastic_Strain" type="bool"
          value="true"/>
          <Parameter name="Von_Mises_Stress" type="bool" value="true"/>
         </ParameterList>
</ParameterList>
<ParameterList name="Output2">
        <Parameter name="Output File Type" type="string"
        value="ExodusII"/>
        <Parameter name="Output Filename" type="string"
        value="ViscoplasticNeedlemanFullyPrescribedTension_NoFlaw"/>
        <Parameter name="Output Frequency" type="int" value="1"/>
        <ParameterList name="Output Variables">
          <Parameter name="Max_Von_Mises_Stress" type="bool"
          value="true"/>
          <Parameter name="Min_Von_Mises_Stress" type="bool"
          value="true"/>
          <Parameter name="Global_Kinetic_Energy" type="bool"
          value="true"/>
        </ParameterList>
</ParameterList>
```

The file ViscoplasticNeedlemanFullyPrescribedTension_NoFlaw.e contains all quantities with respect to Output1 and stores those for at the global coordinates. The file ViscoplasticNeedlemanFullyPrescribedTension_NoFlaw.h contains global quantities and computed quantities defined by the user with respect to Output2. When the user starts the interactive Python interface, the following steps should be done at first.

```
import netCDF4
```

```
import numpy as np
import matplotlib.pyplot as plt
nc = netCDF4.Dataset(
'ViscoplasticNeedlemanFullyPrescribedTension_NoFlaw.h')
```

First, load the necessary modules and then the exodus file (here, the output of a Peridigm test is chosen). The data is loaded then to the field `nc`. This field shows all subfields containing different variables by using

```
nc.variables
```

It gives an overview about all information stored in the output file and the size of the fields. Now, the information of the fields can be stored in python variables. In this example, we want to plot the stresses over the strains. In the input file the maximum and minimum von Mises stresses have been defined to be written to the output. The maximum stresses are at the first and the minimum stresses at the second position. This position is the same in the output field `vals_glo_var`. This field can be accessed as

```
var = nc.variables['vals_glo_var']
```

Now all maximum stresses are in [:,0] and all minimum stresses are in [:,1]. These can be written to distinct fields.

```
vm_max = var[:,0]
vm_min = var[:,1]
```

Last, we need the time steps. The way of getting the values is the same.

```
time_steps = nc.variables['time_whole']
dt = time_steps[:]
```

Since the example has a constant strain rate it can be calculated easily by

```
eng_strain_Y = dt * 0.001 /1.0e-8
```

Then the plot can be shown with the following commands

```
plt.plot(eng_strain_Y, vm_max, eng_strain_Y, vm_min)
plt.ylabel("Max/Min von Mises Stress")
plt.show()
```

For further plot options look at
`https://matplotlib.org/users/pyplot_tutorial.html` and a detailed explanation is given at
`http://johntfoster.github.io/posts/extracting-exodus-information-with-netcdf-python.html`.

2.2.2. How to include Peridigm and the Trilinos environment in Eclipse

Eclipse is a powerful editor for C++ or Java projects. Peridigm and its libraries can also be imported into the editor. The steps are listed as follows.

- File → Import → C/C++ → Existing Code as Makefile Project
- Existing Code Location: Browse to ∼/<Path to Peridigm directory>
- Languages: C++
- Choose 'Linux GCC' as toolchain
- Highlight the Peridigm project in the project manager window pane and right-click on 'properties'
  - C/C++ Build: untick Generate Makefiles automatically
  - C/C++ Build → Environment: Add the environmental variables
    * Name: BOOST_DIR and Value: ∼/<Path to Boost directory>
    * Name: HDF5_DIR and Value: ∼/<Path to HDF5 directory>
    * Name: NETCDF_DIR and Value: ∼/<Path to NETCDF directory>
    * Name: TRILINOS_DIR and Value: ∼/<Path to Trilinos directory>

    Leave PWD and CWD as default
  - C/C++ General → Paths and Symbols: Add the include and libary paths of all external libraries. Choose the following include paths:
    * ∼/<Path to Boost directory>/include/boost
    * ∼/<Path to HDF5 directory>/include
    * ∼/<Path to NETCDF directory>/include
    * ∼/<Path to Trilinos directory>/include
    * ∼/<Path to OpenMPI directory>/include

    And choose the following library paths:
    * ∼/<Path to Boost directory>/lib
    * ∼/<Path to HDF5 directory>/lib64
    * ∼/<Path to NETCDF directory>/lib64
    * ∼/<Path to Trilinos directory>/lib
    * ∼/<Path to OpenMPI directory>/lib64

- Peridigm can be build in Eclipse and all changes are incorporated when the respective file is saved before building the project.

2.2.3. Doxygen support for Peridigm

The doxygen support shows the structure and descriptions of the code. Doxygen can be downloaded as a source file or can be installed directly from the GIT repository, if GIT is installed. The following lines show how to install doxygen with GIT support.

```
git clone https://github.com/doxygen/doxygen.git
cd doxygen
mkdir build
cd build
cmake -G \dq Unix Makefiles\dq ..
make
make install
```

Then the doxygen executable is linked and available as a name via the shell. The doxygen usage is very simple. The actual Peridigm code is located in the `src` directory. Enter the source directory and execute Doxygen.

```
doxygen -g <Doxygen configuration file name>
```

The option `-g` implies to generate a configuration file and its file name can be set arbitrarily. If the name isn't set by the user, Doxygen will create it automatically. Now, the configuration file can be adapted to fit the desired output. Here, the following options are recommended to change.

```
PROJECT_NAME           = \dq Peridigm\dq
OUTPUT_DIRECTORY       = <Path to output directory>
CREATE_SUBDIRS         = YES
ALWAYS_DETAILED_SEC    = YES
INLINE_INHERITED_MEMB  = YES
FULL_PATH_NAMES        = YES
EXTRACT_ALL            = YES
SOURCE_BROWSER         = YES
INLINE_SOURCES         = YES
RECURSIVE              = YES
```

All options are explained more detailed in the Doxygen manual and also in the configuration file. Then, the configuration file is executed by

```
doxygen <Doxygen configuration file name>
```

A HTML and Latex folder are created under the output directory path. The latex script refman.pdf can be created by executing the make file in the folder by `make`. Otherwise, a DVI, PS or PDF can be created by a Latex editor. The HTML folder contains all information about the header and classes. The files can be viewed in a common browser, which is able to display HTML.

Own damage models

Peridigm is programmed in C++. All the essential vectors and matrices are stored as pointer. It must be noted that no Voigt notation is used. Therefore, the stress and strain matrices are stored in a vector of length 9. The file Peridigm_DamageFactory allows the definition of the material. Here, the name in the .xml datasheet and the corresponding C++ file are defined.

3. *Peridigm* - Documentation of Implementations

3.1. How to document

- list all edited files
- list all edited regions within the files
- explain changes

3.2. Implemented damage models

3.2.1. Energy based damage model

3.2.1.1. Changed files

```
./src/core/Peridigm.cpp
./src/core/Peridigm.hpp
./src/core/Peridigm_ModelEvaluator.cpp
./src/core/Peridigm_ModelEvaluator.hpp
./src/damage/Peridigm_DamageModelFactory.hpp
./src/damage/Peridigm_EnergyReleaseDamageModel.cpp
./src/damage/Peridigm_EnergyReleaseDamageModel.hpp
./src/materials/Peridigm_ElasticMaterial.cpp
./src/materials/Peridigm_Material.cpp
./src/materials/Peridigm_Material.hpp
```

3.2.1.2. Documentation of changes

Peridigm.cpp

The following changes had been made.

```
damageModelFieldId = fieldManager.getFieldId(PeridigmField::NODE,
   PeridigmField::VECTOR, PeridigmField::TWO_STEP, "Damage_Model_Data");
auxiliaryFieldIds.push_back(damageModelFieldId);
```

The damage model field is defined. It stores the dilation of all nodes. If defined in Peridigm.cpp it can be used to synchronize the data between cores.

```
threeDimensionalMothership =
Teuchos::rcp(new Epetra_MultiVector(*threeDimensionalMap, 11));
damageModelVal = Teuchos::rcp((*threeDimensionalMothership)(9), false);
// Damage Model data which has to be synchronized
```

The "damageModelVal" vector is a so called mothership vector. It is used to synchronize the data. It is of length node time three. It includes the dilation, the bulk modulus divided by the weighted volume and the shear modulus divided by the weighted volume.

```
for(blockIt = blocks->begin() ; blockIt != blocks->end() ; blockIt++){
  if (blockIt->getMaterialModel()->Name() == "Elastic"){
     damageModelVal->PutScalar(0.0);
     blockIt->
        importData(*damageModelVal, damageModelFieldId,
```

```
        PeridigmField::STEP_NP1, Insert);
      }
}
```

The “damageModelFieldId” data is set to zero for all cores.

```
    modelEvaluator->updateDilatation(workset);
```

The model evaluator is started. Here, the Peridigm_ModelEvaluator.cpp is called and the dilatation is updated.

```
for(blockIt = blocks->begin() ; blockIt != blocks->end() ; blockIt++){
    if (blockIt->getMaterialModel()->Name() == "Elastic"){
      blockIt->importData(*damageModelVal, damageModelFieldId,
      PeridigmField::STEP_NP1, Insert);
    }
}
```

The dilation is exported to the global synchronization vector.

```
for(blockIt = blocks->begin() ; blockIt != blocks->end() ; blockIt++){
    if (blockIt->getMaterialModel()->Name() == "Elastic"){
      blockIt->importData(*damageModelVal, damageModelFieldId,
      PeridigmField::STEP_NP1, Insert);
    }
}
```

The dilation is copied back to the cores. At each core the dilation for each node and its neighbors exists.

Peridigm.hpp

The following changes had been made.

```
//! Global vector for damage model data
Teuchos::RCP<Epetra_Vector> damageModelVal;
```

The code defines a global synchronization vector. This vector is needed to synchronize the dilation between the cores.

```
int damageModelFieldId;
```

The field Id is defined to be usable anywhere in Peridigm.cpp.

Peridigm_ModelEvaluator.cpp

The following changes has been made.

```
void
PeridigmNS::ModelEvaluator::updateDilatation
  (Teuchos::RCP<Workset> workset) const
```

An extra routine has been defined. The updateDilatation routine works only if Elastic material is used. The reason is, that only the linear Peridynamic solid models calculate the dilation. Therefore, the data structure only exists if such material is used. It can be extended to other isotropic ordinary Peridynamic material if needed.

Peridigm_ModelEvaluator.hpp

The following changes had been made.

```
void updateDilatation(Teuchos::RCP<Workset> workset) const;
```

The updateDilatation routine is defined.

Peridigm_DamageModelFactory.cpp

The following changes had been made.

```
#include "Peridigm_EnergyReleaseDamageModel.hpp"
```

Include the damage model data files.

```
else if(damageModelName == "Critical Energy")
damageModel =
  Teuchos::rcp( new EnergyReleaseDamageModel(damageModelParams) );
```

Include the damage model if the option is set in the input deck.

```
invalidDamageModel += ", must be \"Critical Stretch\",
  \"Time Dependent Critical Stretch\", \"Interface Aware\"
  or \"Critical Energy\".\n";
```

Extended error log to print the damage model options for the user.

Peridigm_EnergyReleaseDamageModel.cpp

It is a new routine. It includes the routine "initialize" and "computeDamage". Based on the dilation at each node (ownId and neighborId) calculated in Peridigm_Material.cpp the energy of each bond is calculated. If it exceeds a defined limit the bond fails and the bondDamage value is set to 1. Due to the synchronization the bond damage in both bond directions can be calculated identically.

Peridigm_EnergyReleaseDamageModel.hpp

It is a new routine. It defines all global values as data manager Ids or the influence function as well as the callable sub routines.

Peridigm_ElasticMaterial.cpp

The following changes had been made.

```
int m_horizonFieldId = fieldManager.getFieldId(PeridigmField::ELEMENT,
  PeridigmField::SCALAR, PeridigmField::CONSTANT, "Horizon");
m_fieldIds.push_back(m_horizonFieldId);
```

This is done to get a valid field ID for the horizon field at the specific core. If this is not done, the included "Peridigm_ModelEvaluator.cpp" routine will throw an error.

Peridigm_Material.cpp

The following changes had been made.

```
void
PeridigmNS::Material::computeDilatation
```

The routine has been added. It is basically the compute_dilation routine from the material_utilities.cxx. The difference is that the horizon is not constant. In the end for each node the following values are stored.

```
damageModel[3*p] = *theta;
damageModel[3*p+1] = BM / (*m);
damageModel[3*p+2] = SM / (*m) ;
```

The dilatation *theta, the bulk modulus BM divided by the weighed volume *m and the shear modulus SM divided by the weighed volume *m.

All this data is needed to determine the energy of the bond in both directions identically.

Peridigm_Material.hpp

The following changes had been made.

```
virtual void
computeDilatation(
   const int numOwnedPoints,
   const int* ownedIDs,
   const int* neighborhoodList,
   const double BM,
   const double SM,
   PeridigmNS::DataManager& dataManager) const;
```

An additional routine has been added to calculate the dilation in the Model_Evaluator.cpp.

3.3. Implemented time integration methods

3.3.1. Numerical damping added to Velocity-Verlet

3.3.1.1. Theory

The theory for the Velocity-Verlet solver with damping is derived in the following. The equations to compute position and velocity at the next time step without damping are given by

$$\begin{aligned} x_{n+1} &= x_n + \Delta t\, v_n + \frac{1}{2}\Delta t^2\, a_n \\ v_{n+1} &= v_n + \frac{1}{2}\Delta t\,(\, a_{n+1} + a_n\,)\ . \end{aligned} \tag{3.1}$$

x denotes the position, Δt the time increment, v the velocity, a the acceleration and subscript n the time step number. The Velocity-Verlet algorithm with damping introduces an additional parameter γ connected to the damping term and the velocity can be described by

$$v_n = \frac{x_{n+1} - x_{n-1}}{2\,\Delta t}\ . \tag{3.2}$$

Starting now with the Taylor expansion series

$$\begin{aligned} x_{n+1} &= x_n + \Delta t\, v_n + \frac{1}{2}\Delta t^2\, a_n + \frac{1}{6}\Delta t^3 \dot{a}_n + O(\Delta t^4) \\ x_{n-1} &= x_n - \Delta t\, v_n + \frac{1}{2}\Delta t^2\, a_n - \frac{1}{6}\Delta t^3 \dot{a}_n + O(\Delta t^4) \end{aligned} \tag{3.3}$$

and adding both equations gives the Verlet algorithm

$$x_{n+1} = 2\,x_n - x_{n-1} + \Delta t^2 a_n + O(\Delta t^4)\ . \tag{3.4}$$

Expressing the accerlation in terms of force F and mass m allows to introduce a damping force G.

$$x_{n+1} = 2\,x_n - x_{n-1} + \frac{\Delta t^2}{m}(\, F_n - G_n\,) + O(\Delta t^4) \tag{3.5}$$

The damping force G is defined by a damping coefficient γ and the velocity

$$G(v_n) = \gamma\, v_n = \gamma\, \frac{x_{n+1} - x_{n-1}}{2\,\Delta t}\ . \tag{3.6}$$

Setting eq. (3.5) back by one time step leads to

$$x_n = 2\,x_{n-1} - x_{n-2} + \frac{\Delta t^2}{m}(\, F_n - \gamma\, v_n\,) + O(\Delta t^4)\ . \tag{3.7}$$

By adding eq. (3.5) and eq. (3.7) the resulting equation reads

$$x_{n+1} + x_n = 2\,x_n + x_{n-1} - x_{n-2} + \frac{\Delta t^2}{m}(\,F_n - v_n + F_{n-1} + v_{n-1}\,) + O(\Delta t^4)\,. \quad (3.8)$$

Reformulating the equation, using the velocity interpolation for v_n and v_{n-1}, shown in eq. (3.2) and setting the equation one time step forward leads to the velocity integration for the Velocity-Verlet algorithm with damping

$$v_{n+1} = \frac{1}{1 + \gamma\frac{\Delta t}{2\,m}}\,[\,v_n\,(\,1 - \gamma\,\frac{\Delta t}{2\,m}\,) + \frac{\Delta t}{2}\,(\,a_{n+1} + a_n\,)\,]\,. \quad (3.9)$$

The time integration for the position stays the same as for Verlet-Velocity without damping. The parameter γ is defined in Peridigm in the Verlet solver list

```
<ParameterList name="Verlet">
<Parameter name="Numerical Damping" type="double" value="0.01"/>
```

The value should be chosen with respect to the material parameters in the input file.

3.3.1.2. Changed files

```
./src/core/Peridigm.cpp
```

3.3.1.3. Documentation of changes

Peridigm.cpp

The following changes have been made.

```
double numericalDamping = 0.0;
if(verletParams->isParameter("Numerical Damping")){
  numericalDamping = verletParams->get<double>("Numerical Damping");
}
```

If the parameter for numerical damping is defined in the input file, it will be read.

```
  if(verletParams->isParameter("Numerical Damping")){
      for (int i = 0; i < a->MyLength(); ++i) {
              (*a)[i] = (*a)[i] / (1 + numericalDamping * dt2
              / (*density)[i/3]);
              (*v)[i] = (*v)[i] * (1 - numericalDamping * dt2
              / (*density)[i/3]) / (1 + numericalDamping * dt2
              / (*density)[i/3]);
      }
  }
```

The so-called mothership vectors are updated with the respective coeffcients for the step $v_{n+\frac{1}{2}}$.

```
    if(verletParams->isParameter("Numerical Damping")){
        for (int i = 0; i < a->MyLength(); ++i) {
                (*a)[i] = (*a)[i] / (1 + numericalDamping * dt2
                / (*density)[i/3]);
        }
    }
```

The mothership vector is updated with the respective coeffcients for the step v_{n+1}.

Appendices

A. This document

A.1. Repository

This document is part of the PeriDoX repository. The complete repository can be found at:

https://github.com/PeriDoX/PeriDoX

A.2. Typesetting

This document was originally typeset using the documentclass `dlrreprt` from the DLR-internal RM-LaTeX package.

The RM-LaTeX package is not publicly available. Therefore, this document is compatible with a bootstrap-version of the documentclass, called `bootstrap_dlrreprt`. `bootstrap_dlrreprt` class is part of this repository.

The compilation is performed with `pdflatex` with the following options:

```
pdflatex --shell-escape -synctex=1 -interaction=nonstopmode %source --
    extra-mem-top=60000000
```

The bibliography is compiled with `biber`. The glossary must be compiled with makeindex or, for Windows, the included batch-script may be used. The keyword index is created automatically.

The general compilation order is:

`pdflatex` → `biber` → `makeindex` → `pdflatex` → `pdflatex`

BSD Documentation License